5.3 Horn formulas

In order to display human-level intelligence, a computer must be able to perform at least some modicum of logical reasoning. Horn formulas are a particular framework for doing this, for expressing logical facts and deriving conclusions.

The most primitive object in a Horn formula is a *Boolean variable*, taking value either `true` or `false`. For instance, variables x, y, and z might denote the following possibilities.

$$\begin{aligned} x &\equiv \text{the murder took place in the kitchen} \\ y &\equiv \text{the butler is innocent} \\ z &\equiv \text{the colonel was asleep at 8 pm} \end{aligned}$$

A *literal* is either a variable x or its negation $\overline{x}$ ("NOT x"). In Horn formulas, knowledge about variables is represented by two kinds of *clauses*:

1. *Implications*, whose left-hand side is an AND of any number of positive literals and whose right-hand side is a single positive literal. These express statements of the form "if the conditions on the left hold, then the one on the right must also be true." For instance,

   $$(z \wedge w) \Rightarrow u$$

   might mean "if the colonel was asleep at 8 pm and the murder took place at 8 pm then the colonel is innocent." A degenerate type of implication is the *singleton* "$\Rightarrow x$," meaning simply that x is `true`: "the murder definitely occurred in the kitchen."

2. Pure *negative clauses*, consisting of an OR of any number of negative literals, as in

   $$(\overline{u} \vee \overline{v} \vee \overline{y})$$

   ("they can't all be innocent").

Given a set of clauses of these two types, the goal is to determine whether there is a consistent explanation: an assignment of `true`/`false` values to the variables that satisfies all the clauses. This is also called a *satisfying assignment*.

The two kinds of clauses pull us in different directions. The implications tell us to set some of the variables to `true`, while the negative clauses encourage us to make them `false`. Our strategy for solving a Horn formula is this: We start with all variables `false`. We then proceed to set some of them to `true`, one by one, but very reluctantly, and only if we absolutely have to because an implication would otherwise be violated. Once we are done with this phase and all implications are satisfied, only then do we turn to the negative clauses and make sure they are all satisfied.

In other words, our algorithm for Horn clauses is the following greedy scheme (*stingy* is perhaps more descriptive):

```
Input:  a Horn formula
Output:  a satisfying assignment, if one exists

set all variables to false
```

```
while there is an implication that is not satisfied:
  set the right-hand variable of the implication to true

if all pure negative clauses are satisfied:  return the assignment
else:  return ''formula is not satisfiable''
```

For instance, suppose the formula is

$$(w \wedge y \wedge z) \Rightarrow x, \;\; (x \wedge z) \Rightarrow w, \;\; x \Rightarrow y, \;\; \Rightarrow x, \;\; (x \wedge y) \Rightarrow w, \;\; (\overline{w} \vee \overline{x} \vee \overline{y}), \;\; (\overline{z}).$$

We start with everything `false` and then notice that x must be `true` on account of the singleton implication $\Rightarrow x$. Then we see that y must also be `true`, because of $x \Rightarrow y$. And so on.

To see why the algorithm is correct, notice that if it returns an assignment, this assignment satisfies both the implications and the negative clauses, and so it is indeed a satisfying truth assignment of the input Horn formula. So we only have to convince ourselves that if the algorithm finds no satisfying assignment, then there really is none. This is so because our "stingy" rule maintains the following invariant:

> If a certain set of variables is set to `true`, then they must be `true` in *any* satisfying assignment.

Hence, if the truth assignment found after the *while* loop does not satisfy the negative clauses, there can be no satisfying truth assignment.

Horn formulas lie at the heart of Prolog ("programming by logic"), a language in which you program by specifying desired properties of the output, using simple logical expressions. The workhorse of Prolog interpreters is our greedy satisfiability algorithm. Conveniently, it can be implemented in time linear in the length of the formula; do you see how (Exercise 5.32)?

5.4 Set cover

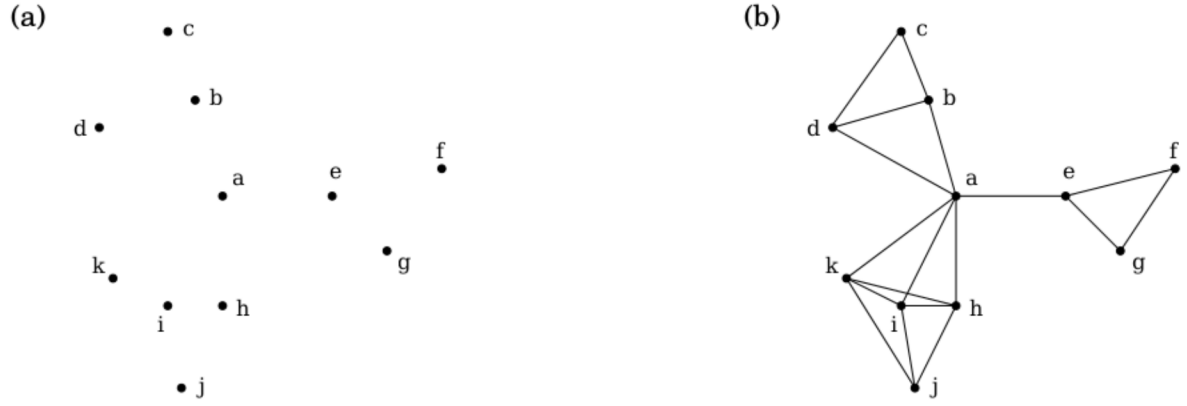
The dots in Figure 5.11 represent a collection of towns. This county is in its early stages of planning and is deciding where to put schools. There are only two constraints: each school should be in a town, and no one should have to travel more than 30 miles to reach one of them. What is the minimum number of schools needed?

This is a typical *set cover* problem. For each town x, let S_x be the set of towns within 30 miles of it. A school at x will essentially "cover" these other towns. The question is then, how many sets S_x must be picked in order to cover all the towns in the county?

> SET COVER
>
> *Input:* A set of elements B; sets $S_1, \ldots, S_m \subseteq B$
>
> *Output:* A selection of the S_i whose union is B.
>
> *Cost:* Number of sets picked.

(In our example, the elements of B are the towns.) This problem lends itself immediately to a greedy solution:

Figure 5.11 (a) Eleven towns. (b) Towns that are within 30 miles of each other.



Repeat until all elements of B are covered:

Pick the set S_i with the largest number of uncovered elements.

This is extremely natural and intuitive. Let's see what it would do on our earlier example: It would first place a school at town a, since this covers the largest number of other towns. Thereafter, it would choose three more schools—c, j, and either f or g—for a total of four. However, there exists a solution with just three schools, at b, e, and i. The greedy scheme is not optimal!

But luckily, it isn't too far from optimal.

Claim *Suppose B contains n elements and that the optimal cover consists of k sets. Then the greedy algorithm will use at most $k \ln n$ sets.*[2]

Let n_t be the number of elements still not covered after t iterations of the greedy algorithm (so $n_0 = n$). Since these remaining elements are covered by the optimal k sets, there must be some set with at least n_t/k of them. Therefore, the greedy strategy will ensure that

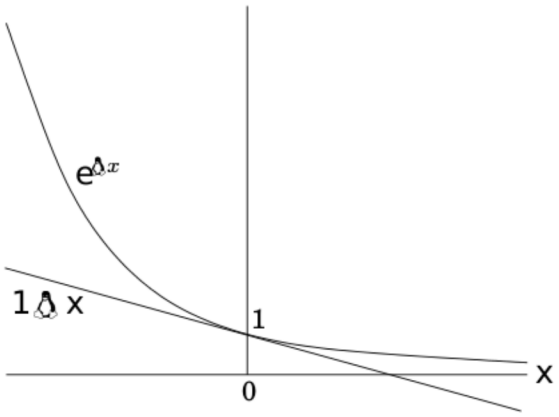
$$n_{t+1} \leq n_t - \frac{n_t}{k} = n_t\left(1 - \frac{1}{k}\right),$$

which by repeated application implies $n_t \leq n_0(1 - 1/k)^t$. A more convenient bound can be obtained from the useful inequality

$$1 - x \leq e^{-x} \text{ for all } x, \text{ with equality if and only if } x = 0,$$

which is most easily proved by a picture:

[2] ln means "natural logarithm," that is, to the base e.



Thus

$$n_t \leq n_0\left(1-\frac{1}{k}\right)^t < n_0(e^{-1/k})^t = ne^{-t/k}.$$

At $t = k \ln n$, therefore, n_t is strictly less than $ne^{-\ln n} = 1$, which means no elements remain to be covered.

The ratio between the greedy algorithm's solution and the optimal solution varies from input to input but is always less than $\ln n$. And there are certain inputs for which the ratio is very close to $\ln n$ (Exercise 5.33). We call this maximum ratio the *approximation factor* of the greedy algorithm. There seems to be a lot of room for improvement, but in fact such hopes are unjustified: it turns out that under certain widely-held complexity assumptions (which will be clearer when we reach Chapter 8), there is provably no polynomial-time algorithm with a smaller approximation factor.